Name

stdio – standard buffered input/output package

Syntax

#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;

Description

The functions described in section 3s constitute a user-level buffering scheme. The in-line macros `getc` and `putc`(3s) handle characters quickly. The higher level routines `gets, fgets, scanf, fscanf, fread, puts, fputs, printf, fprintf, fwrite` all use `getc` and `putc;` they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type FILE. The `fopen`(3s) subroutine creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant 'pointer' NULL (0) designates no stream at all.

An integer constant EOF (–1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard input/output package must include the header file <stdio.h> of pertinent macro definitions. The functions and constants mentioned in sections labeled 3S are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: `getc, getchar, putc, putchar, feof, ferror, fileno.`

VAX Only

On VAX machines, the GFLOAT version of *libc* is used when you use the `cc`(1) command with the **–Mg** option, or you use the `ld`(1) command with the **–lcg** option. The GFLOAT version of *libc* must be used with modules compiled with `cc`(1) using the **–Mg** option.

Also note that neither the compiler nor the linker `ld`(1) can detect when mixed double floating point types are used, and the program may produce erroneous results if this occurs on VAX machines.

System V Compatibility

This library contains System V compatibility features that are available to general ULTRIX programs. For a discussion of how these features are documented, and how to specify that the System V environment is to be used in compiling and linking your programs, see `intro`(3).

Diagnostics

The value EOF is returned uniformly to indicate that a FILE pointer has not been initialized with `fopen`, input (output) has been attempted on an output (input) stream, or a FILE pointer designates corrupt or otherwise unintelligible FILE data.

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default and attempts to do this transparently by flushing the output whenever a `read`(2) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard I/O routines but use `read`(2) themselves to read from the standard input.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to `fflush`(3s) the standard output before going off and computing so that the output will appear.

Files

/lib/libc.a
/usr/lib/libcg.a (VAX only)

See Also

open(2), close(2), read(2), write(2), fread(3s), fseek(3s), ferror(3s), fclose(3s), fopen(3s)

Name

ctermid – generate file name for terminal

Syntax

#include <stdio.h>

char *ctermid(s)
char *s;

Description

The `ctermid` subroutine generates the pathname of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to `ctermid`, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least `L_ctermid` elements. The pathname is placed in this array and the value of *s* is returned. The constant `L_ctermid` is defined in the <stdio.h> header file.

NOTE

The difference between `ctermid` and `ttyname`(3) is that `ttyname` must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while `ctermid` returns a string (`/dev/tty`) that will refer to the terminal if used as a file name. Thus `ttyname` subroutine is useful only if the process already has at least one file open to a terminal.

See Also

ttyname(3)

cuserid (3s)

Name

cuserid – get character login name of the user

Syntax

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

Description

The `cuserid` subroutine generates a character-string representation of the login name of the owner of the current process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least `L_cuserid` characters; the representation is left in this array. The constant `L_cuserid` is defined in the <stdio.h> header file.

Return Value

If the login name cannot be found, `cuserid` returns a NULL pointer; if *s* is not a NULL pointer, a null character (\0) will be placed at s[0].

In POSIX mode, if *s* is not a NULL pointer, *s* is the return value.

Environment

When your program is compiled using the POSIX environment, `cuserid` returns the name associated with the effective userid of the calling process. When compiled in the BSD or System V environments, it returns the name associated with the login activity on the controlling terminal, if any. Otherwise, it returns the same as in the POSIX environment.

See Also

getlogin(3), getpwent(3)

Name

fclose, fflush – close or flush a stream

Syntax

#include <stdio.h>

fclose(*stream*)
FILE **stream*;

fflush(*stream*)
FILE **stream*;

Description

The `fclose` routine causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed. The `fclose` routine is performed automatically upon calling `exit`.

The `fflush` routine causes any buffered data for the named output *stream* to be written to that file. If *stream* is NULL, all open output streams are flushed. The stream remains open.

Diagnostics

These functions return EOF if buffered data cannot be transferred to an output stream.

Environment

If not called in POSIX mode, these functions return EOF if *stream* is not associated with an output file. In POSIX mode, if *stream* is associated with an input file, the file pointer is positioned following the last byte read from that *stream*.

See Also

close(2), fopen(3s), setbuf(3s)

Name

ferror, feof, clearerr, fileno – stream status inquiries

Syntax

```
#include <stdio.h>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

fileno(stream)
FILE *stream;
```

Description

The `ferror` function returns nonzero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by `clearerr`, the error indication lasts until the stream is closed.

The `feof` function returns nonzero when end of file is read on the named input *stream*, otherwise zero.

The `clearerr` function resets both the error and EOF indicators on the named *stream*.

The `fileno` function returns the integer file descriptor associated with the *stream*, see `open`(2).

These functions are implemented as macros; they cannot be redeclared.

See Also

open(2), fopen(3s)

Name

fgetpos, fsetpos – save and restore stream position

Syntax

#include <stdio.h>

int fgetpos (*stream, pos*)
FILE ***stream*;
fpos_t ***pos*;

int fsetpos (*stream, pos*)
FILE ***stream*;
fpos_t ***pos*;

Description

The `fgetpos` function stores the current position of *stream* in *pos*.

The `fsetpos` function restores *stream* to the position returned by an earlier `fgetpos` call.

Return Value

If successful, the return value is zero; on failure, a nonzero value is returned and errno is set to the appropriate value.

See Also

fseek(3s)

Name

fopen, freopen, fdopen – open a stream

Syntax

#include <stdio.h>

FILE **fopen (filename, type)*
char **filename, *type*;

FILE **freopen (filename, type, stream)*
char **filename, *type*;
FILE **stream*;

FILE **fdopen (fildes, type)*
int *fildes*;
char **type*;

Description

The `fopen` routine opens the file named by *filename* and associates a *stream* with it. The `fopen` routine returns a pointer to the FILE structure associated with the *stream*.

The *filename* points to a character string that contains the name of the file to be opened.

The *type* is a character string having one of the following values:

"r"	Open for reading
"w"	Truncate or create for writing
"a"	Append; open for writing at end of file, or create for writing
"A"	Append with no overwrite; open for writing at end-of-file, or create for writing
"r+"	Open for reading and writing
"w+"	Truncate or create for reading and writing
"a+"	Append; open or create for reading and writing at end-of-file
"A+"	Append with no overwrite, open or create for update at end-of-file

The letter "b" can also follow r, w, or a. In some C implementations, the "b" is needed to indicate a binary file, however, it is not needed in ULTRIX. If "+" is used, the "b" may occur on either side, as in "rb+" or "w+b".

The `freopen` routine substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. The `freopen` routine returns a pointer to the FILE structure associated with *stream*.

The `freopen` routine is typically used to attach the preopened *streams* associated with **stdin**, **stdout** and **stderr** to other files.

The `fdopen` routine associates a *stream* with a file descriptor. File descriptors are obtained from `open, dup, creat,` or `pipe`(2), which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many of the

Section 3s library routines. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening `fseek` or `rewind`, and input may not be directly followed by output without an intervening `fseek`, `rewind`, or an input operation which encounters end-of-file.

When a file is opened for append with no overwrite (that is when type is "A" or "A+"), it is impossible to overwrite information already in the file. The `fseek` routine may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

Return Value

The `fopen` and `freopen` routines return a NULL pointer on failure.

Environment

SYSTEM_V

When your program is compiled using the System V environment, append with no overwrite is specified by using the "a" or "a+" type string, and the "A" and "A+" type strings are not allowed.

POSIX

In the POSIX environment, the "a" and "a+" strings, and the "A" and "A+" strings specify append with no overwrite.

See Also

creat(2), dup(2), open(2), pipe(2), fclose(3s), fseek(3s).

fread(3s)

Name

fread, fwrite – buffered binary input/output

Syntax

#include <stdio.h>

size_t fread(*ptr, size, nitems, stream*)
void **ptr*;
size_t *size, nitems*;
FILE **stream*;

size_t fwrite(*ptr, size, nitems, stream*)
void **ptr*;
size_t *size, nitems*;
FILE **stream*;

Description

The `fread` function reads into a block beginning at *ptr*, *nitems* of data of the size *size* (usually sizeof **ptr*) from the named input *stream*. It returns the number of items actually read.

If *stream* is **stdin** and the standard output is line buffered, then any partial output line will be flushed before any call to `read`(2) to satisfy the `fread`.

The `fwrite` function appends, at most, *nitems* of data of the size *size* (usually sizeof **ptr*) beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

Return Value

The `fread` and `fwrite` functions return 0 upon end of file or error.

See Also

read(2), write(2), fopen(3s), getc(3s), gets(3s), printf(3s), putc(3s), puts(3s), scanf(3s)

Name

fseek, ftell, rewind – reposition a file pointer in a stream

Syntax

#include <stdio.h>

int fseek(*stream, offset, ptrname*)
FILE ***stream**;
long *offset*;
int *ptrname*;

long ftell(*stream*)
FILE ***stream**;

void rewind(*stream*)
FILE ***stream**;

Description

The `fseek` function sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value SEEK_SET, SEEK_CUR, or SEEK_END.

The `fseek` function undoes any effects of `ungetc`(3s).

The `ftell` function returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes and is the only foolproof way to obtain an *offset* for `fseek`.

The `rewind` (stream) function is equivalent to `fseek` (stream , 0L, 0, SEEK_SET), except that no value is returned.

Return Value

The `fseek` function returns –1 for improper seeks, otherwise 0.

See Also

lseek(2), fopen(3s)

Name

getc, getchar, fgetc, getw – get character or word from stream

Syntax

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

Description

The `getc` function returns the next character from the named input *stream*.

The `getchar` function is identical to `getc` (stdin).

The `fgetc` function behaves like `getc`, but is a genuine function, not a macro. It may be used to save object text.

The `getw` function returns the next word (in a 32-bit integer on a VAX-11 or MIPS machine) from the named input *stream*. It returns the constant EOF upon end of file or error, but since that is a good integer value, *feof* and `ferror`(3s) should be used to check the success of `getw`. The `getw` assumes no special alignment in the file.

Restrictions

Because it is implemented as a macro, `getc` treats a stream argument with side effects incorrectly. In particular, 'getc(*f++);' doesn't work as expected.

Diagnostics

These functions return the integer constant EOF at end of file or upon read error.

A stop with message, 'Reading bad file', means an attempt has been made to read from a stream that has not been opened for reading by `fopen`.

See Also

fopen(3s), fread(3s), gets(3s), putc(3s), scanf(3s), ungetc(3s)

Name

gets, fgets – get a string from a stream

Syntax

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
FILE *stream;
```

Description

The `gets` routine reads a string into *s* from the standard input stream **stdin**. The string is terminated by a newline character, which is replaced in *s* by a null character. The `gets` routine returns its argument.

The `fgets` routine reads *n*–1 characters, or up to a newline character, whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. The `fgets` routine returns its first argument.

Restrictions

The `gets` routine deletes a newline, while `fgets` keeps it.

Diagnostics

The `gets` and `fgets` routines return the constant pointer NULL upon end of file or error.

See Also

ferror(3s), fread(3s), getc(3s), puts(3s), scanf(3s)

Name

printf, fprintf, sprintf – formatted output conversion

Syntax

#include <stdio.h>

int printf(*format* [, *arg*] ...)
char **format*;

int fprintf(*stream*, *format* [, *arg*] ...
FILE **stream*;
char **format*;

BSD Environment

char *sprintf(*s*, *format* [, *arg*] ...)
char **s*, *format*;

System V and POSIX Environments

int sprintf(*s*, *format* [, *arg*] ...)
char **s*, *format*;

Description

The `printf` function places output on the standard output stream, `stdout`. The `fprintf` subroutine places output on the named output *stream*. The `sprintf` subroutine places output in the string *s*, and appends the null terminator '\0' to the end of the string.

The first argument controls how each of these functions converts, formats, and prints the other arguments. The first argument is a character string that contains two types of objects, characters and conversion specifications. These functions copy characters that appear in the first argument to the output stream. Conversion specifications cause these functions to convert the next succesive argument and send the formatted argument to the output stream.

You introduce conversion specifications using the percent sign (**%**). Following the **%**, you can include:

- Zero or more flags, which modify the meaning of the conversion specification.
- An optional minus sign (–), which specifies left adjustment of the converted value in the indicated field.
- An optional digit string that specifies a field width. If the converted value has fewer characters than the field width, `printf` pads the value with blanks. By default, `printf` pads the value on the left. If the conversion string specifies the value is left-justified, `printf` pads the value on the right. If the field width begins with a zero, `printf` pads the values with zeros, instead of blanks.
- An optional period (.), which separates the field width from the next digit string.
- An optional digit string specifying a precision. The precision controls the

number of digits that appear after the radix character, exponential and floating-point conversions. Precision also controls the maximum number of characters that are placed in the converted value for a string.

- The character **h** or **l** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** corresponds to an integer or longword integer argument. You can use an uppercase **L** or a lowercase **l.**
- A character that indicates the type of conversion to be applied.

A field width or precision can be an asterisk (*), instead of a digit string. If you use an asterisk, you can include an argument that supplies the field width or precision.

The flag characters and their meanings are as follows:

– The result of the conversion is left-justified within the field.

+ The result of a signed conversion always begins with a sign (+ or –).

blank
If the first character of a signed conversion is not a sign, `printf` pads the value on the left with a blank. If the blank and plus sign (+) flags both appear, `printf` ignores the blank flag.

The result has been converted to a different format. The value is to be converted to an alternative form.

For **c, d, s,** and **u** conversions, this flag has no effect.

For **o**
conversions, this flag increases the precision to force the first digit of the result to be a zero.

For **x** or **X** conversions, `printf` pads a non-zero result on the left with **0x** or **0X**.

For **e, E, f, g,** and **G** conversions, the result always contains a radix character, even if no digits follow that character. (A radix character usually appears in the result of these conversions only if a digit follows it.)

For **g** and **G** conversions, `printf` does not remove trailing zeros from the result.

The conversion characters and their meanings are as follows:

dox Convert the integer argument to decimal, octal, or hexadecimal notation, respectively.

f Convert the floating point or double precision argument to decimal notation in the style *[–]ddd.ddd*, where the number of *d*s following the radix character is equal to the precision for the argument. If the precision is missing, `printf` prints six digits. If the precision is explicitly zero, the function prints no digits and no radix characters.

e Convert the floating point or double precision argument in the style *[–]d.ddde±dd*, where one digit appears before the radix character and the number of digits that appear after the radix character is equal to the precision. When you omit the precision, `printf` prints six digits.

g Convert the floating point or double precision argument to style **d**, style **f**, or style **e**. The style `prinf` uses depends on the format of the converted value.

The function removes trailing zeros before evaluating the format of the converted value.

If a radix character appears in the converted value that is followed by a digit, `printf` uses style **d**. If the converted value contains an exponent that is is less than –4 or greater than the precision, the function uses style .BR e . Otherwise, the `printf` function uses style **f**.

c Print the character argument.

s Print the character argument. The `printf` function prints the argument until it encounters a null characters or has printed the number of characters specified by the precision. If the precision is zero or has not been specified, `printf` prints the character argument until it encounters a null character.

u Convert the unsigned integer argument to a decimal value. The result must be in the range of 0 through 4294967295, where the upper bound is defined by MAXUNIT.

i Convert the integer argument to decimal. (This conversion character is the same as **d**.)

n Store the number of characters formatted in the integer argument.

p Print the pointer to the argument. (This conversion character is the same as %08X).

% Print a percent sign (%). The function converts no argument.

A non-existent or small field width never causes truncation of a value. Padding takes place only if the specified field width exceeds the length of the value.

In all cases, the radix character `printf` uses is defined by the last successful call to `setlocale` category `LC_NUMERIC`. If `setlocale` category `LC_NUMERIC` has not been called successfully or if the radix character is undefined, the radix character defaults to a period (.).

International Environment

LC_NUMERIC If this environment is set and valid, `printf` uses the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `printf` uses the international language database named in the definition to determine collation and character classification rules. If `LC_NUMERIC` is defined, its definition supercedes the definition of LANG.

Restrictions

The `printf` function cannot format values that exceed 128 characters.

Examples

To print a date and time in the form Sunday, July 3, 10:02, where *weekday* and *month* are pointers to null-terminated strings use the following function call:

```
printf("%s, %s %d, %02d:%02d",
                    weekday, month, day, hour, min);
```

To print π to 5 decimal places use the following call:

```
printf("pi = %.5f", 4*atan(1.0));
```

Return Values

In the BSD environment, `printf` and `fprintf` return zero for success and EOF for failure. The `sprintf` subroutine returns its first argument for success and EOF for failure.

In the System V and POSIX environments, `printf`, `fprintf`, and `sprintf` return the number of characters transmitted for success. The `sprintf` function ignores the null terminator (\0) when calculating the number of characters transmitted. If an output error occurs, these routines return a negative value.

See Also

ecvt(3), nl_printf(3int), nl_scanf(3int), setlocale(3), putc(3s), scanf(3s), environ(5int)
Guide to Developing International Software

putc(3s)

Name

putc, putchar, fputc, putw – put character or word on a stream

Syntax

#include <stdio.h>

int putc(*c, stream*)
char *c*;
FILE **stream*;

putchar(*c*)

fputc(*c, stream*)
FILE **stream*

putw(*w, stream*)
FILE **stream*;

Description

The `putc` routine appends the character *c* to the named output *stream*. It returns the character written.

The `putchar(c)` routine is defined as `putc` (c, stdout).

The `fputc` routine behaves like `putc`, but is a genuine function rather than a macro.

The `putw` routine appends word (that is, **int**) *w* to the output *stream*. It returns zero. The `putw` routine neither assumes nor causes special alignment in the file.

Restrictions

Because it is implemented as a macro, `putc` treats a stream argument with side effects incorrectly. In particular, 'putc(c, *f++);' doesn't work as expected.

Diagnostics

The `putc`, `putchar`, and `fputc` functions return the constant EOF upon error. The `putw` function returns a non-zero value on error.

See Also

fclose(3s), fopen(3s), fread(3s), getc(3s), printf(3s), puts(3s)

Name

puts, fputs – put a string on a stream

Syntax

#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;

Description

The `puts` subroutine copies the null-terminated string *s* to the standard output stream **stdout** and appends a new line character.

The `fputs` subroutine copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

Restrictions

The `puts` subroutine appends a new line, while `fputs` does not.

See Also

fopen(3s), gets(3s), putc(3s), printf(3s), ferror(3s) fread(3s)

scanf(3s)

Name

scanf, fscanf, sscanf – convert formatted input

Syntax

#include <stdio.h>

int scanf(*format*[, *pointer*] ...)
char **format*;

int fscanf(*stream*, *format* [, *pointer*] ...)
FILE **stream*;
char **format*;

int sscanf(*s*, *format* [, *pointer*] ...)
char **s*, **format*;

Description

Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string, *format,* and a set of *pointer* arguments that indicate where to store the converted input. The `scanf` function reads from the standard input stream *stdin*. The `fscanf` function reads from the named input *stream*. The `sscanf` function reads from the character string *s*.

In the *format* string you specify how to convert the input stream. You may use one or more conversion specifications in a single format string, depending on the number of *pointer* arguments you specify. Conversion specifications are introduced by a percent sign and specify the format of one input field. You may also use spaces, tabs, form feeds, new-line characters, alphabetic characters, and numbers in the format string. The following list describes conversion specifications and the other components of a *format* string:

- Conversion specifications have the following format:

  `%[*][w][l][h][code]`

  Each conversion specification must be introduced by a percent sign. The rest of the conversion specification is optional and has the following purpose:

***** Specifies that an input field in the input string is not read by `scanf`; that is, the function skips the field.

w Specifies the maximum field width.

l Specifies that the variable where the input value is stored is a longword integer or a double-precision variable. The `scanf` function ignores the **l** if the input field is a character string or a pointer.

h Specifies that the variable where the input value is stored is a short integer or floating-point variable. The `scanf` function ignores the **h** if the input field is a character string or a pointer.

type Specifies the conversion code. Possible values for the conversion code are described in the paragraphs that follow.

- Alphabetic characters and numbers that appear inside the *format* string, but not in a conversion specification, specify that `scanf` ignore those characters in the input string.
- The white-space characters in a *format* string that appear outside of a conversion specification normally have no effect on how `scanf` formats data. The exception is when the white space character precedes the **c** conversion code in the *format* string. In this case, the white space causes `scanf` to ignore leading white space in the input field. Normally, `scanf` treats leading white space as part of the input character string for the **c** conversion code.

Each conversion specification in the *format* string directs the conversion of the next input field. The `scanf` function stores the result of each conversion in the *pointer* that corresponds to the conversion specification. Thus, the conversion specification controls how `scanf` converts the first unread input field, and `scanf` stores the result in the first *pointer*. The second conversion specification controls how `scanf` converts the next input field. The `scanf` function stores the result of the second conversion in the second *pointer*, and so on.

You do not include *pointers* for conversion specifications that contain the asterisk character. These specifications cause `scanf` to ignore an input field, so no *pointer* storage is needed.

An input field is defined as a string of non-space characters; it begins at the first unread character and extends to the first inappropriate character or EOF. An inappropriate character is one that is not valid for the value `scanf` is reading. For example, the letter "z" is invalid for an integer value. If the `scanf` function does not reach EOF and encounters no inappropriate characters, the field width is the number of characters specified by *w*. For all conversion codes except left-bracket ([) and **c**, `scanf` ignores leading white space in an input field.

The conversion code controls how `scanf` converts an input field. The data type of a *pointer* that corresponds to a conversion specification must match the conversion code. For example, the *pointer* that corresponds to a **c** conversion code must point to a character variable. The *pointer* that corresponds to a **d** conversion code must point to an integer, and so on. The following list describes the valid conversion codes:

% The input field is a percent sign. The `scanf` function does not move any value to *pointer*.

d D The input field is a decimal integer; the corresponding *pointer* must point to an integer. If you specify **h** , *pointer* can point to a short integer.

u U The input field is an unsigned decimal integer; *pointer* must point to an unsigned integer.

o 0 The input field is octal integer is expected; the corresponding *pointer* must point to an integer. If you specify **h** , *pointer* can be a short integer.

x X The input field is a hexadecimal integer; the corresponding *pointer* must point to an integer pointer. If you specify **h**, *pointer* can be a short integer.

e,f,g The input field is an optionally signed string of digits. The field may contain a radix character and an exponent field begins with a letter **E** or **e**, followed by an optional sign or space and an integer. The *pointer* must point to a floating-point variable. If you specify **l**, *pointer* must point to a double-precision variable.

s The input field is a character string. The *pointer* must point to an array of characters large enough to contain the string and a termination character (\0). The `scanf` function adds the termination character automatically. A white-space character terminates the input field, so the input field cannot contain spaces.

c The input field is a character or character string. The *pointer* must point to either a character variable or a character array.

The `scanf` function reads white space in the input field, including leading white space. To cause `scanf` to ignore white space, you can include a space in front of the conversion specification that includes the **c**.

[The input field is a character string. The *pointer* must point to an array of characters large enough to contain the string and a termination character (\0). The `scanf` function adds the termination character automatically.

Following the left bracket, you specify a list of characters and a right bracket ( ]). The `scanf` function reads the input field until it encounters a character other than those listed between the brackets. The `scanf` function ignores white-space characters.

You can change the meaning of the characters within the brackets by including a circumflex (^) character before the list of characters. The circumflex causes `scanf` to read the input field until it encounters one of the characters in the list.

You can represent a range of characters by specifying the first character, a hyphen (-), and the last character. For example, you can express [0123456789] using [0–9]. When you use a hyphen to represent a range of characters, the first character you specify must precede or be equal to the last character you specify in the current collating sequence. If the last character sorts before the first character, the hyphen stands for itself. The hyphen also stands for itself when it is the first or the last character that appears within the brackets.

To include the right square bracket as a character within the list, put the right bracket first in the list. If the right bracket is preceded by any character other than the circumflex, `scanf` interprets it as a closing bracket.

At least one input character must be valid for this conversion to be considered successful.

i The input field is an integer. If the field begins with a zero, `scanf` interprets it as an octal value. If the field begins with ''0X'' or ''0x, `scanf` interprets it as a hexadecimal value. The *pointer* must point to an integer. If you specify **h**, *pointer* can point to a short integer.

n The `scanf` function maintains a running total of the number of input fields it has read so far. This conversion code causes `scanf` to store that total in the integer that corresponds to *pointer*.

p The input field is a pointer. The *pointer* must point to an integer variable.

In all cases, `scanf` uses the radix character and collating sequence that is defined by the last successful call to `setlocale` category `LC_NUMERIC` or `LC_COLLATE`. If the radix or collating sequence is undefined, the `scanf` function uses the C locale definitions.

International Environment

LC_NUMERIC If this environment is set and valid, `scanf` uses the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `scanf` uses the international language database named in the definition to determine collation and character classification rules. If `LC_NUMERIC` is defined, its definition supersedes the definition of LANG.

Restrictions

You cannot directly determine whether conversion codes that cause `scanf` to ignore data (for example, brackets and asterisks) succeeded.

The `scanf` function ignores any trailing white-space characters, including a newline character. If you want `scanf` to read a trailing white-space character, include the character in the conversion code for the data item that contains it.

Examples

The following shows an example of calling the `scanf` function:

```
int i, n; float x; char name[50];

n = scanf("%d%f%s", &i, &x, name);
```

Suppose the input to the `scanf` function appear as follows:

```
25 54.32E-1 thompson
```

In this case, `scanf` assigns the value 25 to the *i* variable and the value 5.432 to the *x* variable. The character variable *name* receives the value thompson\0. The function returns the value 3 to the *n* variable because it read and assigned three input fields.

The following example demonstrates using the **d** conversion code to cause `scanf` to ignore characters:

```
int i; float x; char name[5];

scanf("%2d%f %*d %[0-9]", &i, &x, name);
```

Suppose the following shows the input to the function:

```
56789 0123 56a72
```

In this case, the `scanf` function assigns the value 56 to the *i* variable and the value

789.0 to the *x* variable. The function ignores the 0123 input field, because the **%*d** conversion specification causes `scanf` to skip one input field. The function assigns 56 to *name*; it reads the first two characters in the last input field and stops at the third character. The letter 'a' is not in the set of characters from 0 to 9.

Return Values

The `scanf` function returns the number of successfully matched and assigned input fields. This number can be zero if the `scanf` function encounters invalid input characters, as specified by the conversion specification, before it can assign input characters.

If the input ends before the first conflict or conversion, `scanf` returns EOF. These functions return EOF on end of input and a short count for missing or invalid data items.

Environment

In POSIX mode, the **E**, **F**, and **X** formats are treated the same as the **e**, **f**, and **x** formats, respectively; otherwise, the upper-case formats expect double, double, and long arguments, respectively.

See Also

atof(3), nl_scanf(3int), getc(3s), printf(3s), environ(5int)
Guide to Developing International Software

Name

setbuf, setbuffer, setlinebuf, setvbuf – assign buffering to a stream

Syntax

#include <stdio.h>

setbuf(*stream, buf*)
FILE ***stream**;
char ***buf**;

setbuffer(*stream, buf, size*)
FILE ***stream**;
char ***buf**;
int *size*;

setlinebuf(*stream*)
FILE ***stream**;

int setvbuf(*stream, buf, type, size*)
FILE ***stream**;
char ***buf**;
int *type*; **size_t** *size*;

Description

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a new line is encountered or input is read from stdin. The routine `fflush`, may be used to force the block out early. Normally all files are block buffered. For further information, see `fclose`(3s). A buffer is obtained from `malloc`(3) upon the first `getc` or `putc` on the file. If the standard stream **stdout** refers to a terminal it is line buffered. The standard stream **stderr** is always unbuffered.

The `setbuf` routine is used after a stream has been opened but before it is read or written. The character array *buf* is used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered. A manifest constant BUFSIZ tells how big an array is needed:

```
char buf[BUFSIZ];
```

The `setbuffer` routine, an alternate form of `setbuf`, is used after a stream has been opened but before it is read or written. The character array *buf* whose size is determined by the *size* argument is used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

The `setlinebuf` routine is used to change **stdout** or **stderr** from block buffered or unbuffered to line buffered. Unlike `setbuf` and `setbuffer` it can be used at any time that the file descriptor is active.

The `setvbuf` routine may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type*, defined in stdio.h are:

_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered; the buffer will be flushed when a new line is written, the buffer is full, or input is requested.
_IONBF	causes input/output to be completely unbuffered.

If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. The *size* specifies the size of the buffer to be used. The constant BUFSIZ in <stdio.h> is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

A file can be changed from unbuffered or line buffered to block buffered by using `freopen`. For further information, see `fopen`(3s). A file can be changed from block buffered or line buffered to unbuffered by using `freopen` followed by `setbuf` with a buffer argument of NULL.

Restrictions

The standard error stream should be line buffered by default.

The `setbuffer` and `setlinebuf` functions are not portable to non 4.2 BSD versions of UNIX.

See Also

malloc(3), fclose(3s), fopen(3s), fread(3s), getc(3s), printf(3s), putc(3s), puts(3s).

Name

tmpfile – create a temporary file

Syntax

#include <stdio.h>

FILE *tmpfile ( )

Description

The `tmpfile` subroutine creates a temporary file and returns a corresponding FILE pointer. The file will automatically be deleted when all references to the file have been closed. The file is opened for update.

See Also

creat(2), unlink(2), fopen(3s), mktemp(3), tmpnam(3s)

Name

tmpnam, tempnam – create a name for a temporary file

Syntax

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

Description

These functions generate file names that can safely be used for a temporary file.

The `tmpnam` subroutine always generates a file name using the path-name defined as `P_tmpdir` in the <stdio.h> header file. If *s* is NULL, `tmpnam` leaves its result in an internal static area and returns a pointer to that area. The next call to `tmpnam` will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least `L_tmpnam` bytes, where `L_tmpnam` is a constant defined in <stdio.h>; `tmpnam` places its result in that array and returns *s*.

The `tempnam` subroutine allows the user to control the choice of a directory. The argument *dir* points to the path-name of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a path-name for an appropriate directory, the path-name defined as `P_tmpdir` in the <stdio.h> header file is used. If that path-name is not accessible, `/tmp` will be used as a last resort. This entire sequence can be up-staged by providing an environment variable TMPDIR in the user's environment, whose value is a path-name for the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the **pfx** argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

The `tempnam` subroutine uses `malloc`(3) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from `tempnam` may serve as an argument to *free*. For further information, see `malloc`(3). If `tempnam` cannot return the expected result for any reason, that is `malloc` failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

Notes

The `tmpnam` and `tempnam` routines generate a different file name each time they are called.

Files created using these functions and either `fopen` or `creat` are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use `unlink`(2) to remove the file when its use is ended.

Restrictions

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or `mktemp`, and the file names are chosen so as to render duplication by other means unlikely.

See Also

creat(2), unlink(2), fopen(3s), malloc(3), mktemp(3), tmpfile(3s)

ungetc(3s)

Name

ungetc – push character back into input stream

Syntax

#include <stdio.h>

ungetc(*c, stream*)
FILE **stream*;

Description

The `ungetc` routine pushes the character *c* back on an input stream. That character will be returned by the next `getc` call on that stream. The `ungetc` routine returns *c*. One character of pushback is guaranteed in all cases.

The `fseek`(3s) routine erases all memory of pushed back characters.

Diagnostics

The `ungetc` routine returns EOF if it cannot push a character back.

Environment

In POSIX mode, the file's EOF indicator is cleared.

See Also

fseek(3s), getc(3s), setbuf(3s)

Name

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

Syntax

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

Description

The `vprintf`, `vfprintf`, and `vsprintf` routines are the same as `printf`, `fprintf`, and `sprintf`, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `varargs`(3).

Examples

The following demonstrates how `vfprintf` could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
       .
       .
       .
/*
 *   error should be called like
 *        error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 * separately declared because of the definition of varargs.
 */
va_dcl
{
    va_list args;
    char *fmt;
```

```
    va_start(args);
    /* print out name of function causing error */
    (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
    fmt = va_arg(args, char *);
    /* print out remainder of message */
    (void)vfprintf(stderr, fmt, args);
    va_end(args);
    (void)abort( );
}
```

See Also

varargs(3)